Eficiencia de Algoritmo
Introducción y Ejemplos Sencillos

Estructuras de Datos

Docente: Pamela Landero Sepúlveda
p.landero@uandresbello.edu

¿Qué queremos de un algoritmo?

Correctitud:

Dado un problema, debe producir siempre una solución correcta al problema.

Eficiencia :

Debería usar los recursos computacionales disponibles en forma eficiente (memoria, procesador, almacenamiento, etc)

¿Qué otra característica considera importante?

Eficiencia

Medida del uso de los recursos computacionales requeridos por la ejecución de un algoritmo en función del tamaño de las entradas.

T(n): Tiempo empleado para ejecutar el algoritmo con una entrada de tamaño n

- Tiempo!
- Un algoritmo que da un resultado correcto pero demora demasiado, no sirve mucho.
- Si el GPS se demora 40 minutos en encontrar el camino más corto, ni siquiera lo abriría.

¿Cómo medir el tiempo?

El tiempo depende de muchos factores:

- La rapidez del computador que lo ejecuta
- El lenguaje de programación seleccionado
- La calidad del compilador
- La habilidad del programador
- **¿programas que se están ejecutando al mismo tiempo?**

¿Cómo medir el tiempo?

Los expertos en computación tienen una notación para medir el tiempo llamada **“notación asintótica”.**

Este tipo de notación busca dejar fuera los elementos particulares del hardware donde se ejecuta el programa, permitiendo mostrar la complejidad esencial de un algoritmo.

Notación Asintótica

Se usa en ciencia de la computación para describir la complejidad de un algoritmo.

Específicamente, la notación O-grande indica el “tiempo” utilizado por un algoritmo en el escenario del PEOR CASO.

O(1) - Constante

O(1) Describe un algoritmo que siempre se va a ejecutar en el mismo tiempo, sin importar el tamaño del conjunto de datos ingresado.

O(1) - Ejemplo

```
boolean isPrimerElementoNull(String[] datos){
   if(datos[0] == null){
      return true;
   }
   return false;
}
```

O(N) - lineal

O(N) Describe un algoritmo cuyo tiempo de ejecución crece linealmente con respecto al tamaño de los datos.

Típicamente contienen un ciclo que recorre todos los datos.

O(N) - Ejemplo

```
boolean contieneValor(String[] datos, String valor)
  {
   for(int i = 0; i < datos.length; i++){
      if(datos[i] == valor){
         return true;
      }
   }
   return false;
  }
```

$O(N^2)$ - Cuadratico

$O(N^2)$ Representa un algoritmo cuyo tiempo de ejecución crece de acuerdo con el cuadrado del tamaño de los datos.

Típicamente contiene un ciclo anidado dentro de otro ciclo.

$O(N^2)$ - Ejemplo

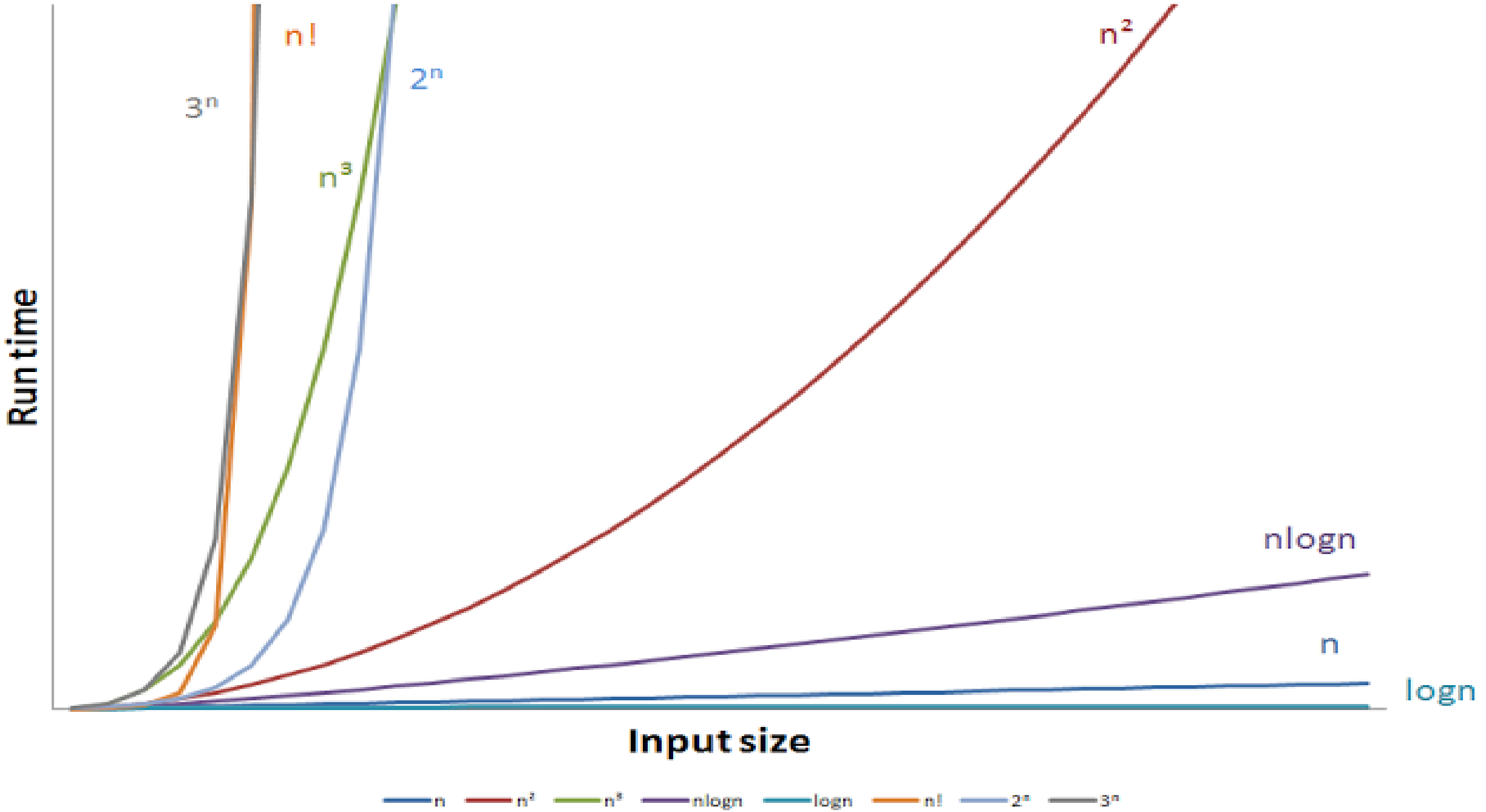
```
boolean ContieneDuplicados(String[] datos)
   {
    for(int i = 0; i < datos.length; i++){
        for(int j = 0; j < datos.length; j++){
            if(i == j){
            continue;
            }
            if(datos[i] == datos[j]){
                return true;
            }
        }
    }
    return false;
   }
```

Funciones orden de eficiencia más habituales

$O(1)$	Orden constante
$O(log\ n)$	Orden logarítmico
$O(n)$	Orden lineal
$O(n\ log\ n)$	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinomio
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

Funciones orden de eficiencia más habituales

- O(1): Complejidad constante. Cuando las instrucciones se ejecutan una vez.
- O(*log n*): Complejidad logarítmica. Esta suele aparecer en determinados algoritmos con iteración o recursión no estructural, ejemplo la búsqueda binaria.
- O(*n*): Complejidad lineal. Es una complejidad buena y también muy usual. Aparece en la evaluación de bucles simples siempre que la complejidad de las instrucciones interiores sea constante.
- O(*n log n*): Complejidad cuasi-lineal. Se encuentra en algoritmos de tipo divide y vencerás como por ejemplo en el método de ordenación quicksort y se considera una buena complejidad. Si *n* se duplica, el tiempo de ejecución es ligeramente mayor del doble.
- O(n^2): Complejidad cuadrática. Aparece en bucles o ciclos doblemente anidados. Si n se duplica, el tiempo de ejecución aumenta cuatro veces.
- O(n^3): Complejidad cúbica. Suele darse en bucles con triple anidación. Si *n* se duplica, el tiempo de ejecución se multiplica por ocho. Para un valor grande de n empieza a crecer dramáticamente.
- O(n^a): Complejidad polinómica ($a > 3$). Si *a* crece, la complejidad del programa es bastante mala.
- O(2^n): Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en subprogramas recursivos que contengan dos o más llamadas internas. ☠


n!
3^n
2^n
n^3
n^2
nlogn
n
logn
Run time
Input size
n
n^2
n^3
nlogn
logn
n!
2^n
3^n

Reglas Asintóticas

Dado el tiempo de ejecución $T_1(n)$ de orden $O(f_1(n))$ y $T_2(n)$ de $O(f_2(n))$. Se cumple lo siguiente:

- Para el tiempo de ejecución de orden $O(\mathbf{c} \cdot f_1(n))$, queda de orden $O(f_1(n))$
- $O(f_1(n)) + O(f_2(n)) = \mathbf{O}(f_1(n) + f_2(n))$ = Dejar el orden dominante (el máximo).
- $O(f_1(n)) * O(f_2(n)) = \mathbf{O}(f_1(n) * f_2(n))$
- $O(f_1(n)) / O(f_2(n)) = \mathbf{O}(f_1(n) / f_2(n))$

Reglas Asintóticas

- Tiempo de ejecución de la ***asignación*** es de orden O(1).
- Tiempo de ejecución de una ***estructura condicional***
  - La condición es de orden O(1).
  - Obtener el orden del conjunto de instrucciones ejecutadas cuando se cumple la condición
  - Obtener el orden del conjunto de instrucciones ejecutadas cuando no se cumple la condición
  - El tiempo de ejecución de la estructura total corresponde al conjunto que tenga el orden dominante
- El tiempo de ejecución de un algoritmo se calcula secuencialmente, sumando el orden de cada estructura. Quedando como resultado final el orden más dominante de todos.

Reglas Asintóticas

- El tiempo de ejecución de una ***estructura repetitiva*** depende del número de veces que se ejecuta el ciclo.
  - Obtener el orden del encabezado del ciclo **O**($f_1(n)$). Por ejemplo si el ciclo recorre todos los datos el orden es O(n) .
  - Obtener el orden del cuerpo del ciclo **O**($f_2(n)$)
  - Por lo tanto, el tiempo de ejecución de la estructura repetitiva completa es del orden **O**($f_1(n) * f_2(n)$)
- La llamada a un procedimiento (o algoritmo) es de O(1).
- Si se requiere determinar el tiempo total de un algoritmo1 que llama a otro algoritmo2, ¿cuánto sería el tiempo de ejecución total del algoritmo1?

Ejemplo 1

```
Ejemplo: Función que utiliza sentencias condicionales

double obtenerCosto(double uf, double x)          O(1)
{
        double costo = 0;                         O(1)
        if (uf < x)                               O(1)
        {
                int y = 2*x;                      O(1)
                costo= y*uf;                      O(1)
        }else{

                int y = 1/2*x;                    O(1)
                costo= y;                         O(1)
        }
        return costo;                             O(1)
}
```

¿Cuál es su resultado?.

Ejemplo 2

Ejemplo: función no recursiva que halla el factorial de un número *n* cualquiera

```
int factorial (int n)                    O(1)
{
        int fact = 1;                    O(1)
        for (int i=n; i>0; i--)          O(n)
        {
                fact = fact*i;           O(1)
        }
        return fact;                     O(1)
}
```

¿Cuál es su resultado?.

Ejemplo 3

Ejemplo: Si el bucle se repite un número fijo de veces.

```
int y, z, k = 10;                    O(1)
for(int i = 0; i < k; i++)           k * O( 1 )
{
      y = y + i;                     O(1)
      z = z + k;                     O(1)
}
```

¿Cuál es su resultado?.

Ejemplo 4

Ejemplo: Dos bucles anidados dependientes de n.

```
for(int i = 0; i < n; i++)                          O(n)
{
      for(int z = 0; z < n; z++)                    O(n)
      {
            if(vector[z] > vector[z + 1])           O(1)
            {
            aux = vector[z];                        O(1)
            vector[z] = vector[z + 1];              O(1)
            vector[z + 1] = aux;                    O(1)
            }
      }
}
```

¿Cuál es su resultado?.

Tenemos O(n) * O(n) * O(1) = $O(n^2)$, complejidad cuadrática.

Ejemplo 5

Ejemplo: Dos bucles anidados, uno dependiente n y otro dependiente del bucle superior.

```
for(int i = 0; i < n; i++)                          O(n)
{
      for(int z = n; z < i; z--)                    O(n)
      {
            if(vector[z] < vector[z - 1])           O(1)
            {
            aux = vector[z];                        O(1)
            vector[z] = vector[z - 1];              O(1)
            vector[z - 1] = aux;                    O(1)
             }
      }
}
```

Cuál es su resultado?.

$O(n) * O(n) * O(1) = O(n^2)$, complejidad cuadrática.

Ejemplo 6

Ejemplo: Bucles donde la evolución de la variable de control es ascendente no lineal.

```
int c = 1;                                          O(1)
while(c < n)                                        O(log n)
{
      if(vector[c] < vector[n])                     O(1)
      {
            aux = vector[n];                        O(1)
            vector[n] = vector[c];                  O(1)
            vector[c] = aux;                        O(1)
      }
            c = c * 2;
}
```

¿Cuál es su resultado?.

O(1) + O(log n) * O(1) = O(log n), complejidad logarítmica.

Ejemplo 7

Ejemplo: Bucles donde la evolución de la variable de control es descendente no lineal.

```
int c = n;                          O(1)
while(c > 1)                        O(log n)
{
        vector[c] = c;              O(1)
        c = c / 2;                  O(1)
}
```

¿Cuál es su resultado?.

O(1) + O(log n) * O(1) = O(log n), complejidad logarítmica.

Ejemplo 8

Ejemplo: Bucles donde la evolución de la variable de control no es lineal, junto a bucles con evolución de variable lineal.

```
int c, x;                                   O(1)
for(int i= 0; i < n; i++)                   O(n)
{
c = i;                                      O(1)
      while(c > 0)                          O(log n)
      {
            x = x % c;                      O(1)
            c = c / 2;                      O(1)
      }
      x = x + 2;                            O(1)
}
```

¿Cuál es su resultado?.

O(1) + O(n) * O(log n) = O(n log n), complejidad cuasi-lineal.

LO IMPORTANTE QUE DEBE SABER :

- Diferenciar un algoritmo de otro, desde el punto de vista de la eficiencia.
- Conocer el funcionamiento de los algoritmos principales de búsquedas y ordenamiento y compararlos según su eficiencia (leer material adjunto de lectura).
- Implementar algoritmos de búsqueda y ordenamiento utilizando estructuras de datos.